FORSCHUNGSZENTRUM JÜLICH GmbH
Zentralinstitut für Angewandte Mathematik
D-52425 Jülich, Tel. (02461) 61-6402

Interner Bericht

Requirements for Automatic Performance Analysis APART Technical Report

Graham D. Riley, John R. Gurd

FZJ-ZAM-IB-9919

November 1999

(letzte Änderung: 10.11.99)

Requirements for Automatic Performance Analysis
APART Technical Report [1]

http://www.fz-juelich.de/apart

Workpackage 1
Requirements for Automatic Analysis Support

G.D. Riley and J.R. Gurd
Centre for Novel Computing,
Department of Computer Science,
University of Manchester,
email:[griley, john]@cs.man.ac.uk

Draft: November 10, 1999

[1] The ESPRIT IV Working Group on *Automatic Performance Analysis: Resources and Tools* is funded under Contract No. 29488

Thanks

The content of this report results from the input and effort of many people. We are particularly grateful to all our partners in the APART Working Group for their contributions during a number of brainstorming sessions held in the past year. One result of these sessions was the production of a questionnaire, which has been distributed to a number of people in the institutions of several of our APART partners. Comments in the completed questionnaires have greatly informed our efforts and we are grateful to all those people who completed and returned forms. Copies of the questionnaire are available on the APART web page (http//:www.fz-juelich.de/apart) and further submissions are still welcome.

Abstract

This report discusses the requirements for automatic performance analysis tools. The discussion proceeds by first examining the nature and purpose of performance analysis. This results in an identification of the sources of performance data available to the analysis process and some properties of the process itself. Consideration is then given to the automation of the process. Many environmental factors affecting the performance analysis process are identified leading to the definition of a scenario space in which users operate. The requirements associated with some scenarios defined by this space are then explored.

Performance is characterised in terms of a number of performance properties associated with low level, hardware-oriented information and also with high level diagnostic statements. Performance properties are defined in terms of the performance data. Performance analysis requires the detection of relevant properties in the execution of an application in the regions of the source code in which they occur along with a statement of their impact on performance. The requirements for defining the data model, detecting properties and locating and quantifying their effect on performance are summarised in a set of high level requirements and requirements derived from these.

The process of automatic performance analysis is summarised as being one of query formulation (describing conditions for the existence of properties etc.) and query optimisation (using suitable strategies to search for properties in performance data which may be gathered on demand).

An object-oriented data model is suggested as a suitable vehicle for representing all possible performance data, and the requirement for formal languages to describe performance properties, and capture the associated dynamic search process for them, is stated.

Table of Contents

1. Introduction

This section provides a summary of the aims of WP1 as expressed in the APART proposal and discussed at the kick-off meeting. Following this, the approach taken in this report to the discussion of requirements for automatic performance analysis is presented and the structure of the rest of the report described.

There have been a number of APART meetings during the last year. The results of these meetings have been a distillation of requirements and the development of a solution to the problem of knowledge representation raised by the requirements. This report presents a discussion of the requirements and the WP2 technical report**,** *Knowledge Specification for Automatic Performance Analysis*, available from the APART web page, presents the technical solution. Note that detailed discussion of related work is presented in the WP2 technical report along with references to the literature.

1.1. Proposal Summary

The aim of APART, as stated in the proposal, is to *promote the efficient use of parallel computers* by enhancing the support for automatic performance analysis provided to application developers.

It is intended that this be achieved by undertaking a thorough analysis of the requirements for automatic performance analysis systems and the issues associated with building and applying them. Results of the project will be integrated in tools being developed by project partners under separate funding.

An important consideration of the project is to investigate the extent of the relationships between the underlying issues of automatic performance analysis in the variety of programming models available to users (current and anticipated) and on the variety of high performance (parallel) platforms (current and anticipated).

The APART proposal states that the requirements analysis will take into account the following points of view:

- Programming interface,
- Application areas,
- Application programmers,
- Available performance monitoring support,
- Future machine designs.

1.2. Approach

The derivation of requirements in this report is first motivated through the examination of the following questions:

- What is the nature of performance analysis?
- What is the purpose of performance analysis?
- What could be automated in performance analysis?

These questions provide the context for the requirements analysis. Following this discussion, a basis for the examination of the requirements for automatic support of performance analysis will be described in terms of a number of *factors* related to the overall environment in which users operate. Selecting a value for each of the factors identifies a *user scenario*. Ideally, the set of scenarios represented should be *complete* in the sense that all users can be associated with a scenario. Some example scenarios are identified and their requirements are explored.

The focus of APART is on *automatic* support for performance analysis, but performance analysis is an integral part of performance optimisation. Analysis is, essentially, a passive activity in the sense that the behaviour of a system (an application code executing on a (parallel) machine) is analysed and reported; no attempt is made to modify the observed behaviour. Attempts to modify performance take place in a subsequent optimisation phase and, clearly, these attempts will be informed by the results of analysis. Note that some optimisation, that made by a compiler for example, will have already taken place as part of an initial analysis, and the analysis-optimisation loop may be repeated a number of times.

A number of ways of describing and analysing the performance of an application written in a specific (parallel) style and executed on a particular parallel architecture have been developed and the detailed requirements of analysis systems should take account of these. This report does not focus on specific requirements which arise from the use of particular programming environments (e.g. OpenMP/HPF/MPI) and from execution on different styles of architecture (e.g. shared memory/distributed memory). Rather, this report seeks to clarify the general requirements for automatic performance analysis and details the infrastructure required to support it. The technical report resulting from

Workpackage 2 presents a solution which satisfies many of the requirements identified here. Implementation issues are the subject of Workpackage 3.

Section 2 contains the discussion of the context for the requirements analysis. Section 3 describes the scenario factors and discusses the requirements arising in some particular scenarios. Section 4.1 summarises the high level requirements identified and Section 4.2 summarises requirements derived from these. Finally, an appendix presents some uni-processor-specific requirements.

2. Context for Requirements Analysis

2.1. What is the nature of performance analysis?

Before discussing requirements for automatic performance analysis it is useful to consider the nature of performance analysis itself.

Traditionally, and in particular in computational science and engineering, the focus is on a particular application code executing on a particular computer hardware, and the task of performance analysis is to explain the run-time behaviour of this hardware/software configuration. As will be seen in later sections, this scenario of a fixed application executing on a fixed (usually dedicated) machine can be generalised in many directions: for example, as the level of source code available for libraries used varies (from none to full), or as the scenario of dedicated resources executing a single application moves to one of distributed, multi-tier client-server architectures and agent-based systems. Even in the above simple scenario, there is a complex environment in which execution takes place, variations in which can dramatically affect the resulting performance. This environment includes, at least, the compiler, the run-time system and any libraries which may be being used. Further, if the resources (including processors, memory and i/o bandwidth) are not dedicated, the environment must include all other tasks executing concurrently with the target application code. Clearly, in such situations, understanding and explaining the observed run-time behaviour is a daunting task. Further complexity is introduced depending on whether application performance is being tuned for a specific number of processors or, as is more generally the case, performance over a range of numbers of processors is of interest (i.e. a scalability analysis is required).

Regardless of the above complexity, some common approaches to understanding run-time performance exist. Two views of performance will be examined. These are:

- the hardware utilisation viewpoint, and
- the relation of performance data to source code viewpoint.

The hardware utilisation viewpoint is reasonably straightforward. The hardware upon which the application code is executing has a number of resources, each of which has a limited capacity. For example, a fixed number of processors, each of which may have a number of functional units such as pipelined floating point units of fixed pipe-depth etc.; an amount of memory; an amount of i/o bandwidth etc. These capacities place a strict upper limit on the performance of any application code executing upon the machine (that which the manufacturer guarantees will not be exceeded). The performance of an application code can therefore be judged, objectively, in terms of the resource utilisation it achieves (often expressed as a set of efficiencies) while performing useful computation (that which is strictly necessary.to solve the application problem at hand).

However, the task of measuring the utilisation of specific components is not trivial. For example, there are problems associated with the accuracy of measurement and with the cost of intrusion into the monitored behaviour. These issues will be present for any performance analysis system, automated or not, as will decisions about the trade-offs between accuracy and cost-of monitoring which are faced in manual performance analysis. As will be seen, however, automated systems may benefit from being able to manage the overall data collection problem involved in performance analysis, in contrast to the small-step-at-a-time process followed in manual methods.

In the hardware utilisation view, the output of performance analysis is usually a set of component utilisation efficiencies. In the context of performance improvement, the aim is to increase utilisation of whatever the user decides are the critical components; for example, increasing the utilisation of processor cycles leading to reduced execution time. This raises the question of how the hardware utilisation figures relate to, and arise in the context of, the rest of the environment - the application source code and libraries etc.- which are compiled into the machine instructions that actually drive the hardware components, under the control of the operating system. It is a sub-set of these environmental factors over which the user can exercise control and thereby affect achieved performance, for example, by changing source code or by changing compiler flags and system environment variables.

This leads to the second view of performance analysis: that of the relationship between performance data and the source code. This question is closely related to the question of the nature of the output required from the task of performance analysis. One further factor affecting the source code relationship view is that of the nature of the user of the results of performance analysis. A human user may have different requirements to those of a (performance improvement) tool.

The output of the performance analysis task may take the following forms:

- raw hardware utilisation data for the entire program,
- raw hardware data related to the source code: for source code regions, individual statements, expressions etc. Such data may be summarised over the entire duration of the program (i.e. over all instances of execution of the region) or for a particular (set of) instances if the region is accessed a number of times during execution. Data may also be provided for individual threads or processes executing a region or summed over all threads/processes.
- Interpreted data: users, whether human or software agents, usually do not wish to manipulate raw data. Rather, some form of interpretation or data filtering takes place. The results of interpretation may take the form of summaries of raw performance data in more informative, often gui-based ways (e.g. graphs and charts), or, at a more abstract level, as diagnostic statements about commonly occurring performance problems, for example, the statement that false sharing on some variable is occurring in a region. Such statements are normally made in the context of a particular implementation of an application (typically, in the form of a set of source code files). In principle, there is no reason why similar diagnostic statements should not also be made at an even more abstract level - that of the algorithms used. These could be given in terms of complexity figures. For example, the number of floating point operations used and the memory requirements.

Some specific examples of low level performance information which is relevant to many modern parallel programming paradigms (e.g. OpenMP, HPF and MPI) and machines are:

- execution time,
- single processor features, such as cache misses (including different types of miss at each level in the cache hierarchy), page faults etc.,
- the time for any redundant computations (often redundant calculations are made to avoid communication),
- time spent waiting by the processor, for which there may be a number of reasons – for example, insufficient parallelism, load imbalance and communication waiting time.

Specific to OpenMP information includes:

- the amount of un-parallelised code,
- synchronisation time: for barriers and locks (including both the cost of executing the synchronisation primitives and any associated waiting time),
- remote access costs associated with loads and stores to addresses not present in “local” memory,
- remote access costs due to poor page placement.

Information specific to HPF includes:

- costs associated with implementing inspector-execute constructs,
- procedure re-map costs.

Information specific to MPI includes:

- the time spent in message passing calls (sends, receives, barriers and global reduction operations),
- the costs incurred where late senders and receivers exist,
- the costs associated with communicating messages.

Statements about false sharing problems and incorrect page placement problems for OpenMP and, in MPI, the recognition of an overloaded master process are examples of more high level, diagnostic performance information. Associated with these is some behaviour which can be described in terms of the more primitive, raw performance data: cache misses, remote accesses and distinct patterns of communication, for example. Identification of such diagnostic information requires that these patterns be detected in the raw performance data.

Performance, then, can be characterised in terms of a number of performance properties, ranging from low level, hardware-oriented information, to higher level diagnostic statements. The existence of these properties must be established and their impact on performance objectively judged. It is to be anticipated that there may be several methods available for checking on the existence of a particular property. These methods may well vary in the amount and cost of measurement required, may establish the existence of the property with differing degrees of confidence and

may determine the impact on performance with different degrees of accuracy. These factors will have a bearing on the process of performance analysis.

One further problem in relating performance information to source code is associated with the increasing gap between the implied abstract machine for which source code is written and the actual raw hardware on which the program executes. This gap is widening at both ends. Machine architectures are becoming more and more complex: many processors now have many functional units of varying pipeline depth with increasingly complex instruction scheduling algorithms. Languages continue to abstract further from machine details. HPF, for example, abstracts away from the underlying message passing implementation on distributed memory machines. Object-oriented languages continue this trend. It is still an open question as to what level of abstract machine should be used to explain performance to human users. For OpenMP, a threads, locks and barriers model is often used. A parallel region with multiple parallel loops (OMPDO statements) in the source code is considered to have an associated entry and exit barrier, and there is also a barrier at the end of each parallel loop. Performance data can be presented for the execution between the two points defined by the region entry and exit barrier, and also for each individual loop. This model cannot always be sustained however, as, in many cases, the loop barriers can be removed without affecting the correctness of the computation, either by compiler analysis or by direct programmer intervention (using a *no barrier* directive) and complete loop-level analysis is no longer possible.

Much of the above discussion is independent of parallel programming language and machine type. This suggests that there are common features of the process of performance analysis which will have specialised implementations for specific languages and machines. For example, the region-based approach to relating performance information to source code is generic but requires specialisation to deal with different types of parallel constructs: OpenMP directives, HPF directives, MPI calls etc. Directive-based languages may also share some common features. This suggests the use of an object-oriented representation notation, supporting inheritance, in which this commonality could be captured as a set of base classes which could be specialised for specific implementations. Use of object-oriented techniques also supports extensibility to future languages and machines.

The types of data which can be gathered to support the analysis task include:

- Statically analysable data associated with the source code.
- Dynamically gathered data from instrumented executions of the program.
- Predicted (analytical) performance data, for example, based on source code analysis and machine and other environmental parameters.
- Data from architecture simulators. Simulators can provide both “exact” performance data and idealised data, for example, for execution with zero costs to memory or with zero communication costs.
- User-supplied data. For example, as provided in assertions to the compiler providing information not available from static analysis; critical code regions on which to concentrate effort, etc.
- Machine parameter data. Including both static information related to the configuration (number of processors, memory and cache details etc.) and data gathered from the execution of specially designed test programs.
- History data, containing all of the above, for versions of the application developed as the performance tuning process develops.
- Potentially, history data for other applications and architectures could also be used to inform the analysis task.

The representation of all the data that it is possible to gather is required.

There are many sources of the data available to an instance of the performance analysis task:

- Previously gathered data (including data from earlier versions of the application in the performance tuning cycle).
- Monitoring data available from hardware counters, instrumented source code and software libraries. This data may be gathered on a per-execution basis but also from executions for which dynamic tuning of gathered data takes place. This may include tuning of both the type of data gathered and the region(s) for which data is gathered.
- The invocation of other existing tools.
- Interpretation by other humans.

Mechanisms for determining which data is to gathered and used during the analysis process are required.

There are well known problems associated with the quantities of data available. These include problems associated with excessive amounts of data generated and those associated with the “sparseness” of sampled data. Techniques to filter this data and reduce the amounts generated by dynamic selection and tuning of request are applicable. Other questions relating to the gathering of performance data are:

- How much time should the system spend analysing the performance? The user should be able to trade fidelity of results against the time required to gather, store and analyse them. Such trade-offs could imply the prediction of certain parameters, to avoid further instrumentation runs of the code to measure them.

2.2. What is the purpose of source code level performance analysis?

In the context of performance tuning, the purposes of performance analysis might be summarised as follows:

- To locate, identify and quantify the major causes of performance loss an application suffers in relation to the application source code. Identification might be in terms of individual primitive performance properties (execution time, wait time, number of cache misses etc.) or in terms of higher level diagnostic statements, which have associated patterns of primitive property data.
- To provide suggestions as to what performance tuning steps could be taken (at source code level), accompanied by some indication of the expected benefit in performance.
- To quantify how well the application is executing on the hardware. This could be done either in terms of hardware utilisation or in terms of how well the application could perform – i.e. by providing a realistic ideal performance for the particular algorithm implemented. Algorithm-level changes could even be suggested.

The question of how well the application is executing is closely related to that of when performance tuning should stop. The performance tuning process can be viewed as an optimisation problem where one seeks to minimise, say, execution time over all possible implementations of an application. The current state-of-the-art in performance tuning appears to be that expert programmers identify potentially beneficial changes which could be made to a program, given an analysis of its current performance. The appropriate changes which they seek to identify come from a list honed by experience, as is the informal order in which they are identified and applied. Any beneficial changes identified are then applied, and the resulting performance noted. This process continues in an ad-hoc manner until no further potentially beneficial changes can be identified. It is clear that a more systematic approach to the optimisation process is required to ensure progress towards the global minimum of the solution space which represents the best possible performance which can be achieved. A more systematic approach would also enable informed trade-offs between, for example, programmer effort and potential performance improvement.

2.3. What could be automated in performance analysis?

In essence, the process of performance analysis consists of a systematic examination of performance data gathered for an application in order to identify performance properties in relation to regions of the application source code. This is a search process.

Automation of this process could occur at two levels:

- Automation of the search process itself. This requires that performance properties be defined in terms of the performance data potentially available and, further, that a process of searching for properties is defined. This can be viewed as a process of query formulation and execution.
- The gathering of data required by the search process can be automated. Many sources may be available to be used to satisfy the (set of) queries raised at any given point in the process, and query planning and query optimisation are required (including the planning and execution of data gathering experiments).

2.4. A Process of Performance Analysis

The performance analysis process will generally not be a completely automatic one. As an example, this section presents a simplified version of a performance analysis process, an overview of which is given in *Figure 1: Overview of an analysis process*. This shows the points at which a user may wish to provide input in order to control the process (duration, fidelity, focus, etc.). This figure also shows internal feed-back (and feed-forward) loops between the components of the process.

An overview of this process and the implicit requirements it entails for automation follows:

1. Accept user input which defines the application and any user defined requirements for the analysis (suitable defaults being provided);
2. Gather static performance related data from the source code and possibly other levels in the compilation process - compiler intermediate code, assembly code and the executable file;
3. Instrument the application at any of the above levels (source to executable);
4. Monitor the execution of the program - multiple executions may be required where limits on the amount of information gathered per-execution exist, for example, where only a fixed number of counters may be active at any

one time, as on the SGI O2000, or where scalability over a range of numbers of processors is to be investigated, or where performance over different data set sizes is of interest;

5. Analyse the static data and monitored data to provide (at least) the information required currently by the user - there is clearly a trade-off between pre-computing all possible data which might be required by the user (for example analysing all sections of code) versus performing only sufficient analysis to support the current user requirements.
6. Make the results of the analysis available to the user in a syntax-directed fashion based on any their requirements (an interactive process).

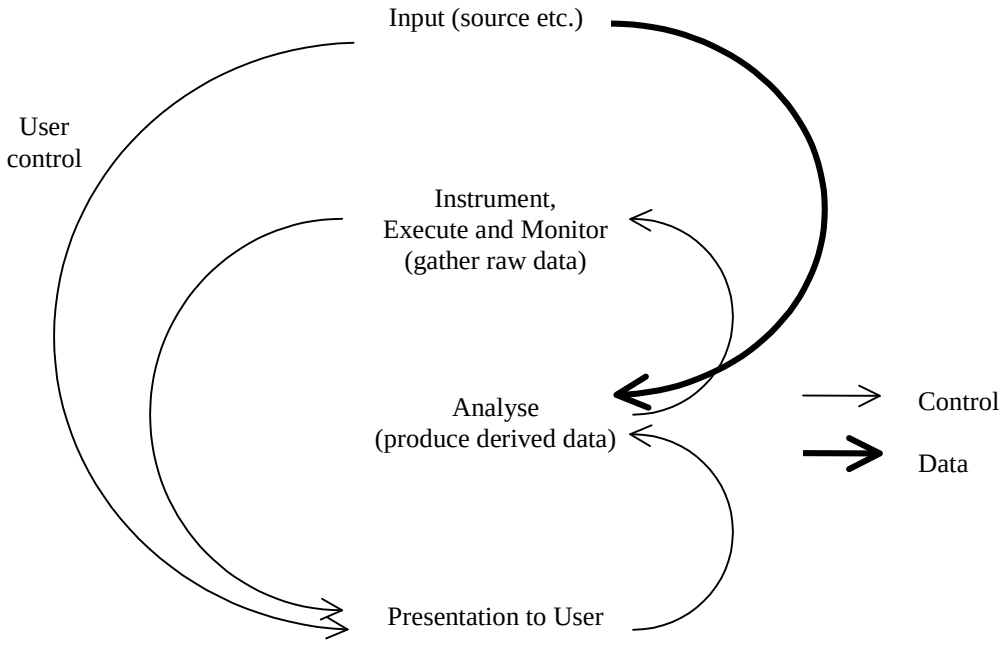

Figure 1: Overview of an analysis process

As indicated in *Figure 1: Overview of an analysis process*, this is not simply a linear process, for example, during analysis further executions might be required to provide extra information. A change of user-view might also require further information gathering. Such requests may take significant amounts of time to satisfy. The user should be able to influence the balance between the response time and the fidelity of the results.

Since the system may be pursuing the identification of several performance properties associated with poor performance in a number of regions at once, it may be able to optimise the number of instrumentation experiments required.

Possible controls on the process to the analyser include:

1. how hard should analyser try? (level-of-difficulty);
2. how long should analyser work at it? (amount of resource);
3. tell me how much better we're doing than last cycle;
4. repeat until not improving (enough);
5. list of things to report.

Possible execution controls:

1. give “before” and “after” profiles;
2. report on overheads or in terms of some other characterisation of performance (e.g. hardware utilisation figures).

3. Scenario Factors

The requirements a user has for support in performance analysis will be affected by many factors linked to the nature of the user's task-in-hand. In this section a number of these factors are described together with the space of user-scenarios which they represent. It is not necessarily the intention that APART should design tools to satisfy all these scenarios. The aim is to delimit the scope of the analysis problem in general and to extract specific requirements by considering certain points (scenarios) in this space.

The following are the major factors associated with the process in which the user is engaged.

What is the nature of the task?
- design-from-scratch,
- parallelisation (of existing code) for performance,
- improvement of already parallelised code,
- etc.

Are there any external processes within which the task is taking place?
- version control systems (from which code would need to be extracted before instrumentation is added, for example),
- etc.

What is the nature of the ``code''?
How is the code composed in terms of source code, library code etc.
- user-source (single or multiple files) vs. source-library vs. binary-library vs. OS calls, etc.

This composition affects the potential for instrumentation and monitoring.

What source language(s) does the program use?
- Fortran 77/90, HPF,
- C,
- C++,
- Java,
- other?

What parallel language style is (to be) used?
- shared memory directives (e.g. OpenMP),
- message passing (e.g. MPI or MPI-2),
- Shmem put/get,
- etc.

What is the nature of the platforms on which the application is to be executed?
- uni-processor,
- multi-processor (homogeneous or heterogeneous resources?),
- distributed memory,
- single-address-space,
- clustered (SMP nodes?),
- etc.

For scientific reproducibility, the hardware configuration and parameters, such as memory capacity, cache sizes, cache line sizes and replacement policies etc., should all be recorded.

What bounds the performance of the application?
- compute bound,
- i/o bound (is parallel i/o used?),
- memory bound (static or dynamic?),
- communication bound,
- other?

Note that performance could be bounded by more than one of the above. Each of these factors represents a different axis in the scenario space.

Environmental factors affecting performance?
- operating system,
- compiler,
- libraries: Numerical, e.g Nag, Nag parallel, IMSL etc., Graphical libraries? Etc.

- libraries: communication (e.g. MPI, PVM, BSP etc.),
- other resources.

For scientific reproducibility the versions of the above resources should be recorded.

What is the nature of the execution environment?
- dedicated or shared resources (processors/memory/IO etc.),
- interactive or batch execution?

What performance metrics are important?
- execution-time
- compile-time
- analysis-time
- code-size
- data-size
- programmer effort (or lack thereof)
- cost of purchasing computer time
- etc.

What is the compilation and build process?
- command line,
- makefile,
- development environment,
- other?

How the executable is built affects what information is available/could be available to be fed into the analysis. For example, could results from one compilation unit be fed forward into subsequent compilations and thus increase the amount/fidelity of the information available to the analysis?
- distinguish compile-time vs. link-time information,
- one-off vs. iterative code-generation vs interpretation,
- etc.

What performance analysis tools are available?
- accurate timers and support libraries,
- hardware monitoring facilities and libraries,
- profilers,
- other tools, such as Apprentice, Dimemas, Pablo, Paradyne, Vampir.

3.1. Example Scenarios

This section gives some examples of user scenarios and briefly discusses their requirements.

3.1.1. A Traditional HPC application

General description:
A N-body simulation code written in Fortran77 for execution in batch mode on a multiprocessor SGI Origin 2000.

Nature of task: parallelisation of existing code
External Processes: none
Nature of code: multiple user-level source files only
Source languages: Fortran 77 only
Parallel language style: OpenMP directives.
Nature of platform: SGI O2000, 16 processors.
Performance bounds: compute bound only.
Environmental factors: (compiler/libraries etc.) standard vendor supplied (details available).
Nature of execution environment: dedicated time on full machine. Batch use only.
Important metrics: execution time, programmer effort. No (unreasonable) limit on memory use.
Compilation and build process: makefile.
Performance analysis tools available: SGI high resolution timers and O2000 hardware counters only.

Discussion: (examples of implications from the above scenario factors).

All source is available, so full instrumentation at source code level is possible. Build is by Makefile so the performance analysis system can take control of this process to compile and instrument multiple experiments under the batch system. The fact that dedicated resources (the entire machine) are available means that reproducibility of timing results is to be expected, allowing multiple executions of the application to be conducted in order to understand its behaviour. Further, scalability experiments across a range of number of processors are possible. Instrumentation of binaries is also possible, as is analytical modelling and prediction. The application is alleged to be compute bound and only execution time is of interest, which suggests that only temporal performance overheads need to be analysed (this does not stop other experiments being performed by the system to check this assumption). Accurate timers and hardware counters are available and the system must be able to interface to these in order to perform its experiments. The nature of the tools available will constrain the nature of the experiments which can be performed.

Some variations arising from minor changes to the scenario might be:

- Given the use of parallelised Nag library routines – usually no source code will be available, so source instrumentation is limited to user code. The extent to which analytical modelling and performance prediction is possible depends on the support provided by the library provider (typically, such support is severely limited). Binary instrumentation is still possible, but reporting might be limited to subroutine level rather than loop nest, statement or expression level.
- Execution on non-dedicated resources – this is a typical situation on single-address-space systems. Even where processors can be reserved, memory cannot usually be partitioned. Such a situation makes "exact" (or confident) performance analysis impossible. Two ways forward might be:
  - the system could make some attempt to indicate the scale of the intrusion by lowering confidence levels in an appropriate fashion when reporting;
  - Reporting could indicate that the analysis took place in a non-dedicated environment and present the analysis in the context of information about that environment during the execution: e.g. by presenting system load profiles, etc.

3.1.2. A Message Passing Application

General description:
Message passing, distributed memory massively parallel machine; batch applications; source with some library calls; parallelisation of sequential code; novice and experienced users; long running as well as short test cases.

Nature of task: improvement of already parallelised code.
External Processes: configuration control system.
Nature of code: multiple source files with some user library code (source available).
Source languages: C.
Parallel language style: MPI
Nature of platform: distributed memory machines (IBM SP2, Cray T3E)
Performance bounds: compute and communication bound
Environmental factors: (compiler/libraries etc.). standard vendor supplied (details available)
Nature of execution environment: batch
Important metrics: execution time and dynamic memory use
Compilation and build process: makefile
Performance analysis tools available: vendor timers and Vampir

Discussion: (examples of implications from the above scenario factors).

This is similar to the previous scenario. The fact that a version control system is used implies that the analyser must be able to extract from and submit source to, it.
The analysis should be able to interface to Vampir to make use of the data this tool can make available. Either, Vampir can be executed once and the data from this then used to inform the analysis process, or Vampir could be invoked to help answer specific questions about specific regions of code which may occur during the analysis. The latter implies that the automatic analyser must know the capabilities of Vampir and be able to make requests of it, i.e., invoke Vampir with parameters and retrieve the results.

Some more specific user requirements associated with this scenario are:

- Batch runs should be supported in two versions: First, the tool itself should be part of the batch job so that a four hour batch job can be used to run multiple experiments. Each experiment might be defined according to previous

results. Second, only the program runs are submitted to batch queues and the tool itself is a separate process submitting jobs individually.

- The final result of the analysis should be a list of performance properties ordered according to their severity.
- The tool should have an explanation facility whereby the user can ask for more information on the detected performance problem.
- The tool should allow the user to guide the search based on his knowledge about the program. For example, the user can specify which region is critical and which property to look for.
- Application programmers are interested in the scalability of the program. Therefore, tools are needed that support automatic analysis of the scalability by running the program on multiple processor configurations.
- Single node performance is crucial on current and future architectures. Detection of performance critical regions should take this into account. Therefore, a loop-level analysis is also required.

3.1.3. An Agent-based application

This final, ficticious, example is given to illustrate an extreme point in the scenario space. Such applications are expected to be of increasing importance in the future.

General description:
A financial services agent is envisaged which will be tasked with finding a "good deal" from those available for purchase on the world wide web, for example, a stock market shares trading agent, where timing and speed of decision are crucial. The agent may be written in Java, for example, but will have to interact and purchase services from a number of databases and possibly other financial services (trend modelling etc.) which may be available from several sources on the web itself (under different conditions of cost and quality of service). Several such databases and services may be accessed and be active concurrently.

Nature of task: concurrent, web-based query satisfaction.
External Processes: the financial markets.
Nature of code: highly distributed and dynamically configured.
Source languages: Java-based but, in general, unknown as the application will consist of remote services.
Parallel language style: remote objects and threads (e.g. as supported in CORBA).
Nature of platform: distributed and unknown, presumably a mixture of sequential and parallel machines.
Performance bounds: unknown at application launch.
Environmental factors: (compiler/libraries etc.). many and unknown at application launch.
Nature of execution environment: non-dedicated and competitive.
Important metrics: execution time and cost bound (costs of purchasing remote services).
Compilation and build process: dynamic via, for example, CORBA mechanisms.
Performance analysis tools available: unknown at application launch time.

Discussion: (examples of implications from the above scenario factors).

The highly dynamic and distributed nature of the environment, including the requirement for run-time configuration, suggests that on-the-fly analysis (and presumably performance tuning) is required. Thus, the analysis must be fast and non-intrusive. Repeatability of results cannot be assumed so multiple experiments cannot be used to investigate behaviour. Since so much is unknown at application launch time, knowledge must be acquired during execution. Analysis for such applications will have to take account of the existence of several sources of services (other software executing on remote machines) and, therefore, mechanisms for the analyser to acquire knowledge of the services and associated performance parameters at run time are required. This, in turn, implies the existence of suitable query protocols.

4. Summary of Requirements

This section summarises the requirements for automatic performance analysis systems. These requirements have arisen, in part, through a consideration of the many factors which determine the scenario in which performance analysis is taking place. Consideration of the analysis process itself has led to the characterisation of performance by a set of properties defined over all the performance-related data which can exist for an application. This includes data derived from the program (both by static analysis and by dynamic instrumentation experiments), data about the machine being used to execute the program, and data derived from other sources, such as the user, analytical modelling and simulation. Analysis proceeds by detecting properties which have an impact on the performance of an application (either at run-time or during post-execution analysis), in relation to both the source code and to the phase of the

execution in which it occurred. The result of an analysis is a quantification of the effect on performance of all significant properties identified.

A major question for the implementation of automatic performance analysers is that of how to manage the search for properties and the associated data collection activity. The process can be summarised as being one of query formulation (describing conditions for the existence of properties etc.) and query optimisation (using suitable strategies to search for properties in performance data which may pre-exist or be gathered on demand).

The requirements are summarised in a set of high level requirements and a set of requirements derived from these. Underlying all these requirements is the need for extensibility to enable the incorporation of future programming models, architectures and performance tools etc.

4.1. High Level Requirements

1. The ability to describe all data pertinent to the process – i.e. provide a data model.
2. The ability to describe the performance properties which are of interest.
3. The ability to detect automatically the presence of performance properties in relation to specific regions of application code (multiple proofs may exist):
   - with some indication of confidence in the detection;
   - with some indication of their contribution to performance loss (their severity);
   - with some indication of the accuracy of the severity;
   - with some indication of the cost of the detection.
4. The ability to use whatever sources of data and information are available to pursue detection.
5. The ability to pursue the search for property detection in a way which minimises the amount of information gathered and which minimises intrusion into the behaviour of the program.
6. The ability to present raw performance data and also to make higher level diagnostic statements about performance to the user (which may be a human or another tool).
7. The ability to quantify the performance loss suffered by the program, and provide an indication of what performance improvement may realistically be expected.

4.2. Derived Requirements

The derived requirements result from considerations of what could be automated in the task of providing support for the high level requirements.

1. The search process itself - a process of query formulation.
2. The data generation and collection process to support the search process, including experiment planning and execution management, where instrumented runs of the program are required -a process of query optimisation.

The following requirements emerge from a consideration of the infrastructure that is required to support automation:

1. Provide an extensible data model in which relevant performance data can be captured (static, dynamic, history, predicted, user-supplied etc.). An object-oriented data model appears to be appropriate. Inheritance can be used to model common features of the data, for example, those arising from source code representation issues. The extensibility of object-oriented models allows for the integration of new programming paradigms and new machines.
2. Provide a formal notation for the description of performance properties (over the data model) - including such factors as: existence conditions, confidence, accuracy, cost of data collection, severity etc. Note that, for many properties, several alternative conditions etc. may be provided, each associated with different proof techniques and costs.
3. Provide a formal language in which to describe the search process - including support for query formulation, dynamic query pursuit and query optimisation.
4. Provide a formal description of interfaces to existing tools to support access to data and query support that the tools can provide during the analysis process - including indications of the cost of queries, where possible, to support optimised query planning.
5. Provide a recommendation system to suggest next steps in both the performance analysis and performance tuning task.

A. Appendix

The scenario space should be able to accommodate all environments in which some form of automatic performance analysis would be useful. For example, the process of tuning sequential code on uni-processors should be accommodated. Also, it can be envisaged that the process of processor (and system) design, when considered as a hardware tuning exercise, could fit into this space. This appendix captures some requirements which point in this direction.

Some Uni-processor Architecture Specific Requirements

The following sections summarise discussions on uni-processor requirements which took place in May 1999 between CNC and Fecit.

- Compiler output analysis. The performance analysis tool could interact with the compiler in order to report the potential benefits of further optimisation which the compiler could do if it had the consent of the user. For example allowing the use of associativity rules, allowing out-of-order execution, planting pre-fetch instructions etc. A user control input over the requirement for reproducibility in results would be required.
- One potential problem is that the relationship between “code” and “performance” is discontinuous. Small changes in code can lead to unexpected large changes in performance.
- Make use of the case where code already contains directives which a sequential compiler is ignoring but which give hints about possible optimisations (no dependency hints, for example, could be used to allow out-of-order execution.). Note: SGI compilers for the Origin 2000, from version 7.2 onwards, do this to some extent.
- The tool could excise ``core'', performance critical, parts of the code (identified during the analysis) and subject them to a detailed *sensitivity* analysis with regard to memory hierarchy behaviour (loop un-rolling, blocking etc.). A test set on which the tool could learn about an architecture would support this activity.
- Analysis reference points. Use estimations of performance assuming zero cost memory accesses and no collisions in pipelines etc.
- Results display. Report the potential benefits of optimisations by showing the user what resource utilisation figures are likely to change and by how much. Order the potential optimisations by effect on the resources of most interest to the user.